VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Navigace v návrhových vzorech

Navigation in design patterns

2014

Michal Ondra

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Michal Ondra**

Studijní program. B2647 Informační a komunikační technologie

Studijní obor· 2612R025 Informatika a výpočetní technika

Téma. Navigace v návrhových vzorech
Navigation in design patterns

Zásady pro vypracování.

1 Přehled současného stavu v oblasti navrhových vzorů.
2. Návrh software pro vyhledávání návrhových vzorů.
3. Implementace navrženého software.
4. Experimenty ověření a vyhodnocení.
5. Závěr

Seznam doporučené odborné literatury·

Erich Gamma, Ralph Johnson, John Vlissides, Richard Helm, Design Patterns. Elements of Reusable Object-Oriented Software, 1994 by Addison-Wesley

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty

Vedoucí bakalářské práce· **prof. RNDr Václav Snášel, CSc.**

Datum zadání. 01.09.2013

Datum odevzdání. 07.05.2014

doc. Dr Ing. Eduard Sojka
vedoucí katedry

prof. RNDr Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7 května 2014

Michal Ondra

Děkuji panu prof. RNDr. Václavu Snášelovi, CSc. za to, že se ujal vedení mé bakalářské práce a jeho cenné rady při konzultacích.

Abstrakt

Cílem této bakalářská práce je uvést čtenáře do problematiky návrhových vzorů a jejich dokumentace. Je popsán vývoj a využití těchto vzorů, struktura pro dokumentaci, uvedeny konkrétní příklady a také popis vytvořené aplikace, kterou lze tyto návrhové vzory dokumentovat, prohlížet a vyhledávat.

Klíčová slova: Návrhový vzor, dokumentace, katalog vzorů, vyhledávání

Abstract

The aim of this bachelor thesis is to introduce reader to design patterns and way to document them. The thesis describes history and use of design patterns, structure of documentation, examples and description of created application for documenting, browsing and searching.

Keywords: Design pattern, documentation, patterns catalog, searching

Seznam použitých zkratek a symbolů

GOF	–	Gang of Four
LINQ	–	Language Integrated Query
OOPSLA	–	Object-Oriented Programming, Systems, Languages and Applications
RTF	–	Rich Text Format
UML	–	Unified Modeling Language
WPF	–	Windows Presentation Foundation
W3C	–	World Wide Web Consortium
XML	–	Extensible Markup Language

Obsah

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Návrhové vzory v softwarovém inženýrství jsou používány pro návrh a implementaci systémů, převážně pomocí objektově orientovaného programování. Použití návrhových vzorů můžeme zajistit nejen správné řešení problémů, které se při vývoji můžou naskytnout, ale také i ušetřit čas, jelikož se aplikují již použité postupy. To může pro firmy, zabývající se vývojem informačních systémů, znamenat možnost optimalizace procesu a ušetřit tak náklady.

Cílem této práce je vytvoření aplikace pro dokumentaci návrhových vzorů. Hlavním důvodem, proč vytvořit takovou aplikaci je ten, že i v dnešní době, po 27 letech od vzniku prvních návrhových vzorů v oblasti softwarového inženýrství, neexistuje žádný jednotný zdroj těchto vzorů. Návrhové vzory se nejčastěji dokumentují formou knih nebo se také dají nalézt na menších webových stránkách, což ale v dnešní době není dostačující. Dalším důvodem je jednotné vyhledávání mezi těmito vzory, které může usnadnit orientaci mezi nimi a jejich výběr.

Práce je rozdělena do několika kapitol. Kapitola 2 pojednává, jak tyto návrhové vzory vznikly, proč je vhodné je používat, dále jejich rozdělení a způsob dokumentace. V kapitole 3 se zaměřím na konkrétní příklady návrhových vzorů. Kapitola 4 pak popisuje detaily a použití samotné aplikace.

2 Uvedení do problematiky návrhových vzorů

V této kapitole je rozebrána historie a přínos návrhových vzorů, jejich rozdělení a struktura dokumentu, kterou se dají vzory popisovat.

2.1 Historie návrhových vzorů

Návrhové vzory pocházejí ze stavebního inženýrství. Vznik je datován do roku 1977, kdy byla vydaná kniha „A Pattern Language: Towns, Buildings, Construction“ [1] architektem Christopherem Alexanderem, která popisuje první návrhové vzory jako návody na řešení složitějších problémů v architektuře, které se často opakovaly. Návrhový vzor byl uveden ve formě předložení určitého problému a následně jeho řešení. Sám Christopher Alexander definoval návrhový vzor následovně:

> *Každý návrhový vzor popisuje problém, který se opakovaně vyskytuje v našem prostředí a popisuje jádro řešení tohoto problému tak, že můžeme použít toto řešení mnohokrát, aniž bychom dělali stejnou věc dvakrát.* [1]

Je nutné však také doplnit, že návrhové vzory nepopisují řešení problému jako jediné takové. Může být spousta dalších technik, které jsou správné a v určitých případech i efektivnější. Návrhový vzor pouze předkládá návod na řešení určitého problému a zaručuje, že právě tímto postupem lze dosáhnout očekávaného výsledku.

Hlavním cílem je zaznamenat techniky expertů na řešení různých problémů při návrhu a implementaci softwarových systémů, které se ukázaly jako správné. Tyto techniky však musí splňovat mimo jiné následující body:

Správnost řešení: návrhový vzor by měl pomáhat problém vyřešit, nikoli naopak. To zahrnuje různé aspekty, jako je výsledná stabilita řešení, relativně malá složitost v porovnání s použitím jiných řešení, optimalizace nebo časová náročnost implementace.

Znovupoužitelnost: návrhový vzor musí řešení popisovat tak, aby se dal použít v neomezeném množství. Není totiž důvod dokumentovat techniku, u které je malá pravděpodobnost, že se ještě někdy použije.

Abstrakce: Návrhový vzor nepopisuje implementaci algoritmů nebo konkrétní situace. Dá se použít v širokém spektru a musí tedy být co nejobecnější.

Netriviálnost: návrhový vzor by neměl popisovat řešení trivialních situací, kterých se dá lépe dosáhnout z vlastních zkušeností.

První výskyt návrhových vzorů v softwarovém inženýrství se uvádí v 90. letech 20. století. V té době se objektově orientované programování stávalo stále více populárním a mnoho expertů v oblasti softwarového inženýrství se začalo touto poměrně novou technikou zabývat. V roce 1987 Ward Cunningham a Kent Beck začali experimentovat s myšlenkou použití návrhových vzorů v programovacím jazyce Smalltalk. Následně své

výsledky později téhož roku prezentovali na konferenci OOPSLA týkající se programování v objektově orientovaných jazycích. Největší úspěch však sklidila takzvaná *skupina čtyř (Gang of Four)*, skládající se z expertů Ericha Gammy, Richarda Helma, Ralpha Johnsona a Johna Vlissidese. Tato skupina vydala v roce 1994 knihu „Design Patterns: Elements of Reusable Object-Oriented Software"[2] představující návrhové vzory v programování široké veřejnosti a uvedení několika základních návrhových vzorů. Kniha znamenala obrovský úspěch a stala se uznávanou. Dalším úspěšným autorem návrhových vzorů je Martin Fowler, který vydal v roce 2003 knihu *Patterns of Enterprise Application Architecture*[3] popisující skupinu návrhových vzorů týkajících se architekturou aplikace.

Návrhové vzory mají mimo jiné tyto výhody a nevýhody:

- Výhody
  - aplikování již použitého řešení – ověřený postup, méně časové náročné, než vymýšlet vlastní řešení
  - zjednodušení návrhu systémů
  - strukturování zdrojového kódu-zlepšuje se čitelnost
  - vzory se dají používat jako jednotka při popisu částí systémů — je jednodušší mluvit o naimplementovaném vzoru, než o detailech zdrojového kódu.
- Nevýhody
  - není vždy zaručena správnost–vzory může napsat kdokoliv.
  - ověření správnosti vzorů se děje až při znovupoužívání.

Návrhové vzory také daly inspirovat vytvoření tzv. anti-vzorů (anti-patterns). Jedná se o popis řešení, u kterého se ze zkušeností ukázalo, že za žádných okolností nemohou přinášet výhody nebo výhody, které přinášejí jsou značně převáženy nevýhodami. Tak byly popsány v knize „AntiPatterns: refactoring software, architectures, and projects in crisis" v roce 1998 Williamem Brownem.

Občas se i samotné návrhové vzory mohou stát anti-vzory. Jak již bylo řečeno, ověření správnosti vzorů probíhá až při znovupoužívání.

2.2 Rozdělení návrhových vzorů

Základní rozdělení dle GOF [2] spadá do tří kategorií, podle toho, jaký problém vzor řeší:

Creational (Vytvářející vzory): jedná se o vzory řešící problémy v oblasti vytváření tříd a objektů, zejména těch které jsou vytvářeny za běhu programu.

Structural (Strukturální vzory): popisují, jakým způsobem strukturovat třídy a objekty. Hlavním cílem je zjednodušit tyto struktury a zvýšit tak přehlednost kódu.

Behavioral (Vzory popisující chování): řeší komunikaci mezi objekty a jejich spolupráci.

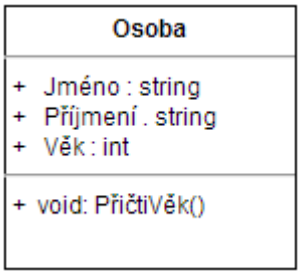
Postupem času vznikly také i jiné kategorie, například Concurrency (vzory řešící problematiku souběhu ve vícevláknových aplikacích) Architectural (vzory týkající se návrhu architektury systému) a jiné.

2.3 Způsob dokumentace

Samotný návrhový vzor se dá popsat jak textově, tak i graficky pomocí tzv. UML diagramů. Pro popis návrhových vzorů se nejčastěji používá statický třídní diagram. Třídní diagramy popisují strukturu tříd a vztahy mezi nimi. Nepopisují interakci mezi třídami z hlediska času.

Samotná třída je znázorněna na obrázku 1. Skládá se ze 3 částí. První představuje název třídy. Druhá část představuje proměnné, u kterých se také uvádí typ a modifikátor přístupu. Třetí část obsahuje seznam metod, které třída obsahuje. Také u nich se uvádí modifikátor a typ. Modifikátory se označují speciálními znaky:

- + označuje modifikátor public
- - označuje modifikátor private
- # označuje modifikátor protected

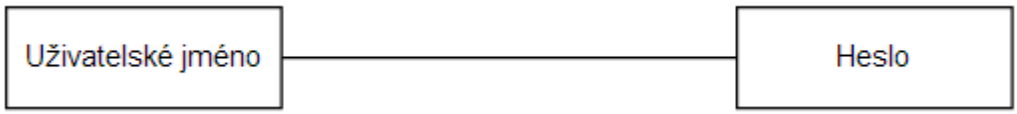


Obrázek 1: Grafický popis třídy

Třídní diagram také popisuje vztahy mezi třídami. Ty jsou důležité pro návrh systému, protože se pomocí nich dá zachytit celá struktura systému. U vazeb agregace a kompozice je také důležitá multiplicita. Ta udává maximální počet objektů každé třídy ve vazbě. Multiplicita může být 1:1, 1:N, M:N nebo také konkrétnější hodnoty, např. 1:2, 0:0..1 atd. Vztahy mezi třídami mohou být následující:

2.3.1 Asociace

Asociace označuje jednoduchou vazbu mezi třídami. Jedna třída používá druhou a naopak.



Obrázek 2: Asociace

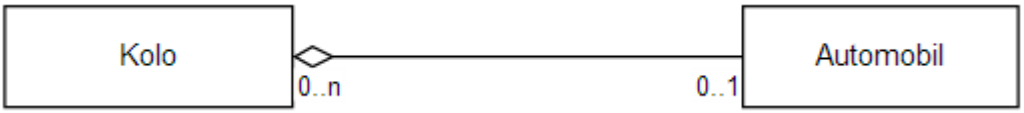
2.3.2 Dědičnost

Dědičnost (generalizace) znázorňuje, že jedna třída (automobil) rozšiřuje druhou (vozidlo). Přebírá její neprivátní metody a proměnné.

Obrázek 3: Dědičnost

2.3.3 Agregace

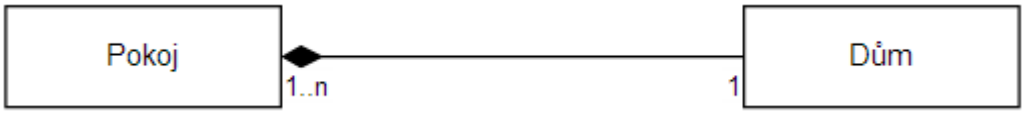
Agregace se používá u složených objektů. Jeden objekt (automobil) vlastní celou kolekci jiných objektů (kolo). Zároveň však kolo může existovat samostatně a nemusí být tak součástí žádného auta.



Obrázek 4: Agregace

2.3.4 Kompozice

Kompozice je zvláštní typ agregace, kdy jeden objekt (dům) obsahuje kolekci jiných objektů (pokoj), zároveň však jednotlivé pokoje nemůžou existovat bez domu, do kterého by patřily.



Obrázek 5: Kompozice

2.4 Struktura návrhových vzorů

Od vzniku návrhových vzorů se objevilo několik názorů, jaká struktura je pro dokumentaci vzorů nejvhodnější, i když se jednotlivé struktury od sebe moc neliší. Já uvedu ty nejpoužívanější, kterou uvedla právě skupina GOF[2]. Celý vzor lze rozdělit do několika sekcí:

Název vzoru: ten musí být vystižný a snadno zapamatovatelný. Většinou se název ponechává v anglickém názvu i v českých literaturách, já však budu uvádět pro lepší pochopení významu vzoru jak název anglický, tak i název český.

Klasifikace: určuje, do které kategorie vzor spadá. Čtenář tak může rychleji pochopit, jaký obecný problém vzor řeší.

Také známý jako: další, často používaná jména. Tato část usnadní vyhledávání v katalogu vzorů, jelikož pokud má vzor další známé názvy, může se stát, že právě použitý název nebude uživatel znát. Samozřejmě, ne vždy má vzor více názvů.

Záměr: krátký popis toho, jaký problém vzor vlastně řeší.

Motivace: příklad konkrétní situace, ve které je dobré vzor použít.

Použitelnost: popisuje, za jakých okolností lze vzor použít a jak tyto okolnosti lze rozpoznat.

Struktura: jedná se o UML diagramy popisující spolupráci mezi objekty.

Účastníci: seznam tříd a objektů, které jsou nezbytné pro implementaci vzoru. Také popisuje jejich úkoly.

Spolupráce: jakým způsobem mezi sebou objekty komunikují, aby dosáhly svého cíle.

Důsledky: výsledek použití návrhového vzoru. Tato část popisuje, jestli vzor s sebou nese nějaké problémy při použití v některých situacích a naopak, jaké výhody jeho použití přináší.

Implementace: popisuje různá úskalí a techniky, kterých je dobré se vyvarovat nebo si dávat pozor, přímo při implementaci návrhového vzoru.

Ukázkový kód: ukázka a popis částí zdrojového kódu, jak by se dal vzor naimplementovat. Nejedná se přímo o algoritmy nebo konkrétní části kódu nějaké aplikace, někdy je však dobré zaznamenat některé části, ve kterých často dochází k různým problémům.

Známá použití: aneb příklady systémů, které tento vzor používají. Jedná se spíše o doplněk, který zvýší věrohodnost vzoru.

Související vzory: seznam vzorů, s kterými je dobré tento vzor kombinovat. Ve většině případů se nikdy vzory nepoužívají samostatně a v určité kombinaci můžou tvořit řešení pro rozsáhlý problém.

3 Příklady návrhových vzorů

Pro hlubší pochopení významu návrhových vzorů uvedu v této kapitole konkrétní příklady od každé ze základních kategorií. Vzory jsou strukturovány do sekcí popsaných v podkapitole 2.4.

3.1 Singleton (Jedináček)

Klasifikace: Structural (Vytvářející vzor)

Záměr: Zajištění toho, aby třída měla pouze jednu instanci.

Motivace: Občas se naskytne situace, kdy chceme zajistit, aby určitá třída měla právě jednu instanci. Příkladem může být okno aplikace nebo internetový server. Právě na takové situace se vzor Singleton zaměřuje. Popisuje řešení, kdy se samotná třída stará o to, aby neměla více instancí.

Použitelnost: Singleton se dá použít, pokud:

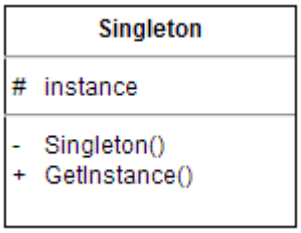
- třída musí mít právě jednu instanci a zároveň také tato instance musí být přístupná v celé aplikaci. Ostatní objekty tak mohou k této instanci snadno přistupovat.
- třída může být rozšiřovaná podle potřeby. Případní potomci budou také mít zajištěnou pouze jednu instanci, avšak je nutné si uvědomit, že tito potomci sdílí s bázovou třídou právě tuto jednu instanci.

Singleton často bývá srovnáván se statickou třídou. Ta má velice podobný výsledek a také určité výhody. Například je snadnější na implementaci, jelikož nemusíme kontrolovat, zda již třída nemá instanci vytvořenou. Jako statická třída je také optimálnější z hlediska rychlosti, jelikož se nevytvářejí žádné instance. Na druhou stranu je však použití vzoru Singleton vhodné, pokud potřebujeme předávat objekt této třídy jako parametr různých metod. Avšak největší výhodu přináší, pokud třída Singleton obsahuje rozsáhlé kolekce objektů. V takovém případě můžeme aplikovat návrhový vzor Lazy load [3], kterým zajistíme, že se tyto objekty načtou až v momentě, kdy jsou potřeba. Další výhodou je možnost třídu rozšiřovat, což statická třída neumožňuje.

Struktura: Diagram třídy Singleton můžeme vidět na obrázku 6

Účastníci:

- Singleton
  - obsahuje instanci sebe sama a statickou metodu, kterou se dá k této instanci přistupovat.
  - zajišťuje inicializaci instance po zavolání konstruktoru. Inicializace se však dá provést pouze jednou.



Obrázek 6: Struktura vzoru Singleton

Spolupráce: Klientské třídy přistupují podle potřeby přímo k instanci třídy Singleton použitím její statické metody k tomu určené.

Důsledky: Použití Singletonu přináší tyto výhody:

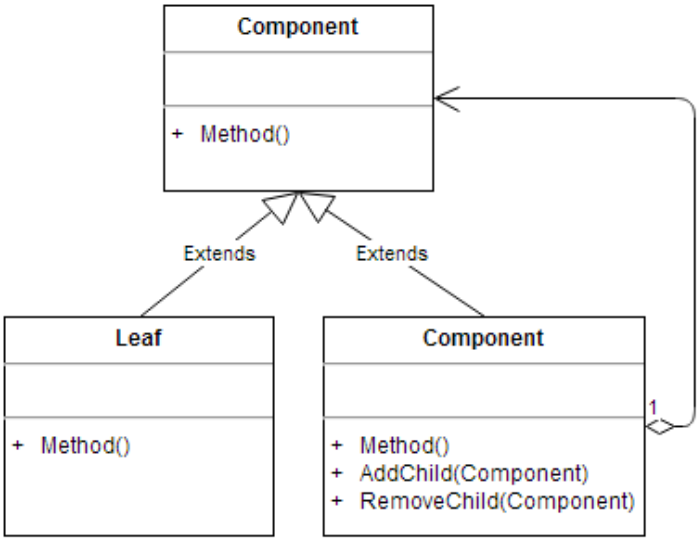
- kontrolovaný přístup k instanci Singletonu. Můžeme tak například zajistit přístup pouze určitým objektům.
- snadná změna třídy. Můžeme například zajistit maximální počet instancí třídy pouhým přepsáním vytvářecí metody.

Implementace: Pro zajištění vytvoření právě jedné instance je nutné zajistit tři základní body:

1. instance třídy Singleton, která se nachází právě v této třídě, jsou nastavené implicitně na hodnotu null. Z důvodu zapouzdření a zajištění rozšiřitelnosti je tato úroveň přístupnosti instance nastavena jako protected.
2. veškeré konstruktory třídy Singleton jsou privátní a tedy přístupné pouze z metod třídy Singleton. Zajistíme tak, že instance třídy Singleton nemohou být vytvářeny z vnějšku.
3. naimplementování statické metody, která vytváří samotnou instanci. Po zavolání zkontroluje, zda již není instance vytvořená. Pokud není, tak zavolá konstruktor. Po vytvoření instance se dá používat pro přístup k této instanci.

Ukázkový kód: příklad implementace vzoru Singleton znázorňuje výpis 6. Objekty pracující s instancí třídy Singleton mohou k této instanci přistupovat pomocí metody GetInstance a dále s ní pracovat jako s normálním objektem.

Související vzory: vzor Singleton se dá použít v kombinaci s mnoha vzory, například již zmíněný Lazy load, kterým zajistíme vytváření instance až ve chvíli, kdy je to nutné. Další častou kombinací se Singletonem je vzor Abstract factory, kdy můžeme dynamicky za běhu programu určit, jakého typu bude instance třídy Singleton.



Obrázek 7: Struktura vzoru Composite

3.2 Composite (Kompozit)

Klasifikace: Structural (Strukturální vzor)

Záměr: skládání objektů do stromové struktury.

Motivace: při programování je často nutné pracovat stromově strukturovánými daty (objekty) stejného nebo podobného typu. Composite zajišťuje skládání těchto objektů do jednoho, pomocí kterého můžeme přistupovat ke všem ostatním.

Použitelnost: composite se dá použít, pokud:

- třídy a objekty přistupující k objektu Composite nerozlišují, jestli se jedná o jednoduchý objekt nebo o kolekci.
- mnoho tříd si jsou velmi podobné tak, že je můžeme skládat do stromové struktury využitím dědičnosti nebo tyto objekty obsahují další objekty stejného typu.

Struktura: diagram tříd návrhového vzoru Composite je znázorněn na obrázku 7

Účastníci:

- Component
  - abstraktní třída nebo rozhraní. Definuje rozhraní pro všechny své potomky.
- Composite
  - její instance představuje kořenový list stromu a další případné potomky, kteří nejsou listy.

  - udržuje reference na všechny své případné potomky.
  - implementuje metody třídy Component. Zároveň může definovat vlastní rozhraní pro své potomky.
- Leaf
  - reprezentuje list v kompozici. Nemá tedy žádné potomky.
  - stejně jako třída Kompozit implementuje metody své bázové třídy.
- Klientské třídy a objekty
  - jedná se o třídy a objekty, které pracují s celou komponentou (instance třídy Composite).

Spolupráce: klientské třídy a objekty přistupují podle potřeby k instanci třídy Composite, která udržuje reference na veškeré své potomky a definuje rozhraní pro operaci s němi.

Důsledky: použití vzoru Composit přináší tyto výhody:

- pomocí jediné instance třídy Composite můžeme pracovat s celou strukturou objektů.
- klientské třídy a objekty nemusí vůbec vědět, jestli instance třídy Composite obsahuje nějaké potomky. Zavoláním jedné metody můžeme vyvolat změnu pro každý objekt v této struktuře.
- pro přidání nového typu objektu (vytvoření nové třidy) do kompozice nemusíme předělávat již naimplementované třídy.

Implementace: často se deklaruje kolekce potomků přímo v abstraktní třídě Component. To je výhodné pouze v případech, kdy je za běhu programu používáno málo listů, jelikož i tyto listy pak mají tuto kolekci, i když je tato kolekce vždý prázdná. Pokud se používá těchto listů ve větším množství, bývá výhodnější tuto kolekci deklarovat ve třídě Composite.

Ukázkový kód: příklad implementace vzoru Composite znázorňuje výpis 2. Abstraktní třída Composite definuje rozhraní pro své potomky Composite a Leaf, metodu Method, která je virtuální, čímž povolíme, aby si jí mohli potomci změnit podle potřeby.

Známá použití: vzor Composite používá skoro každý systém založený na objektech. Nejznámější použití je samotný operační systém Windows, který uchovává adresáře a soubory právě v této struktuře. Kompozit se dá chápat jako každá složka, která obsahuje soubory a případně další složky a list právě jako soubor.

Související vzory: composite bývá často používán s návrhovým vzorem Decorator, čímž můžeme zajistit dynamické rozšiřování komponent.

3.3 Observer (Pozorovatel)

Klasifikace: Behavioral (vzory popisující chování)

Také známý jako: Dependents, Publish-subscribe

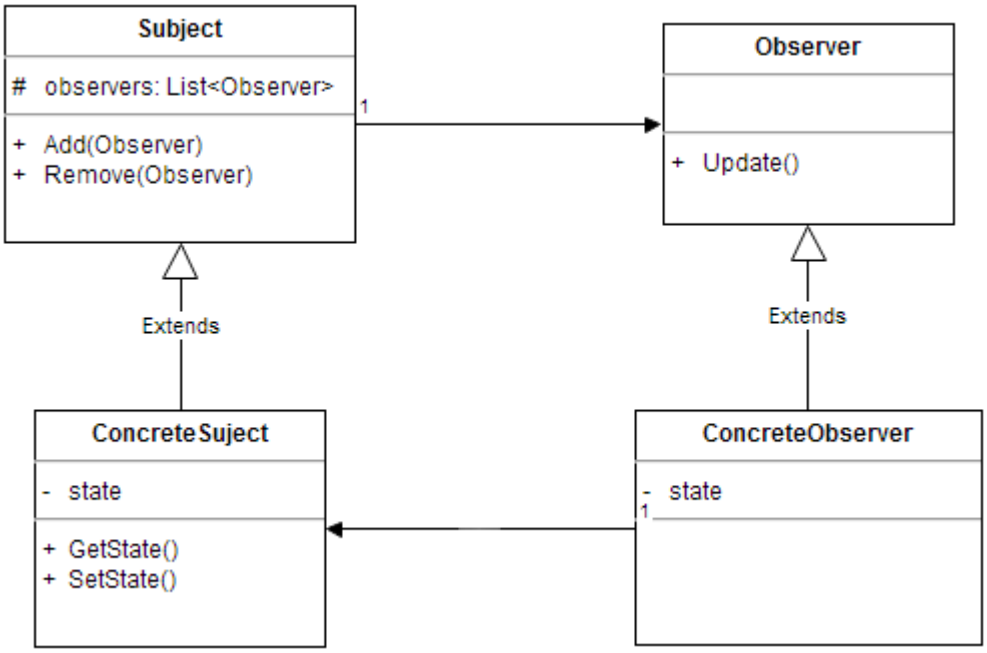
Záměr: definování vazby 1:N pro objekty tak, že jeden objekt informuje své pozorovatele, pokud změní svůj stav.

Motivace: často se stává, že při změně určitého objektu potřebujeme pro zachování konzistence dat změnit i jiné objekty. Právě tímto problémem se zabývá návrhový vzor Observer.

Použitelnost: Observer se dá použít, pokud:

- Změna jednoho objektu vyžaduje změnit i jiné objekty. Zároveň není předem určeno, kolik těchto objektů je.
- Měněný objekt nemusí předem vědět, jakého typu budou zaregistrované objekty.

Struktura: diagram tříd návrhového vzoru Observer je znázorněn na obrázku 8



Obrázek 8: Struktura vzoru Observer

Účastníci:

- Observer

  - abstraktní třída nebo rozhraní. Definuje rozhraní pro všechny objekty, které jsou informovány o změně stavu pozorovaného objektu.
- Subject
  - abstraktní třída nebo rozhraní definující metody pro upozornění pozorovatelů o změně stavu sledovaného objektu.
  - uchovává referenci na všechny své pozorovatele.
  - poskytuje rozhraní pro přidávání a odebírání pozorovacích objektů.
- ConcreteObserver
  - představuje konkretní pozorovatele.
  - uchovává svůj stav, který je konzistentní s pozorovaným objektem.
- ConcreteSubject
  - uchovává seznam referencí na své pozorovatele.
  - pokud změní stav, informuje všechny své pozorovatele.

Spolupráce: Třída ConcreteSubject informuje své pozorovatele kdykoli se změní jeho stav. Ti následně aktualizují svůj stav.

Důsledky: Použití vzoru Observer přináší tyto výhody:

- plně odděluje třídy ConcreteObserver od třídy ConcreteSubject.
- pozorovaný objekt není závislý na počtu svých pozorovatelů.
- pozorovatelé se můžou kdykoli odebírat nebo přidávat.

Implementace: Vzor Observer se dá také použít v případě, kdy pozorovatelé potřebují sledovat více než jeden objekt. Jediná změna se musí provést upravením metody, kterou dává třída Subject vědět o změně tak, aby předával referenci na měněný objekt.

Ukázkový kód: Příklad implementace znázorňuje výpis 3.

Související vzory: Příkladem je návrhový vzor Singleton, který můžeme použít pro třídy Observer a Subject k zajištění jedné globální instance těchto tříd.

4 Vytvoření aplikace pro dokumentaci návrhových vzorů

V rámci této kapitoly popíšu aplikaci, kterou lze návrhové vzory dokumentovat. Tato aplikace má za cíl splňovat tyto body:

1. jednoduché používaní.
2. prostředky pro vytvoření, editaci a smazání vzorů.
3. možnost ukládání a načítání vzorů ze souboru.
4. snadné sdílení návrhových vzorů.
5. rozšířené fulltextové vyhledávání v libovolné kombinaci různých sekcí.
6. podpora více jazyků.

4.1 Použité technologie

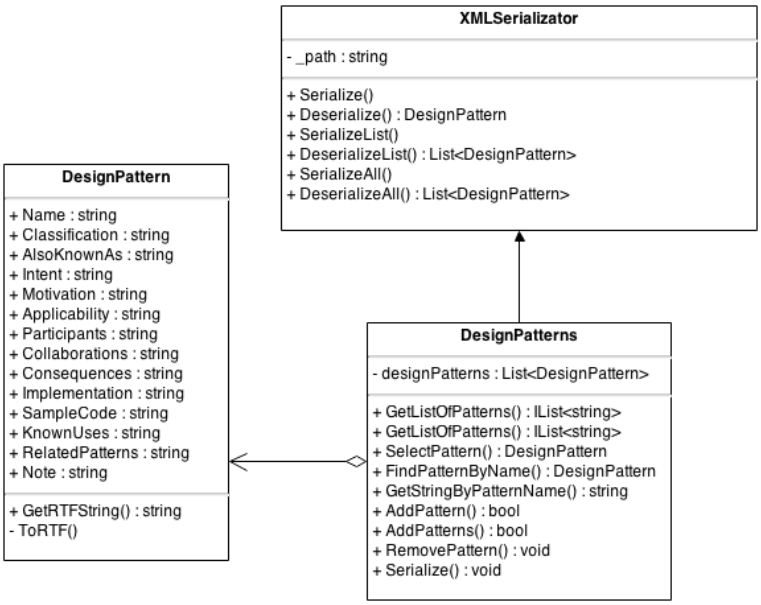
Vytvoření aplikace je dosaženo pomocí programovacího jazyka C# a technologie Windows Forms aplikace. Návrhové vzory se uchovávají do XML souboru pomocí serializace. Výhodou je, že tyto soubory lze snadno přenášet na jiná zařízení, popř. i přečíst pomocí jiné aplikace. Text návrhového vzoru se převádí na formát RTF, pomocí kterého se dá naformátovat pro výstup.

4.1.1 Programovací jazyk C#

Objektově orientovaný jazyk C# byl vytvořen firmou Microsoft v roce 2002 a je přímým nástupcem jazyka C++ [5]. C# však má plno rozdílů, hlavním je ten, že objekty nemohou být hodnotového typu. Tím se více podobá konkurenčnímu jazyku Java od firmy Sun Microsystems z roku 1995. Za roky svého působení se však vyvinul daleko víc a přináší stále nové technologie pro zjednodušení vývoje, např. integrovaný dotazovací jazyk LINQ pro práci s daty obecně (nerozlišuje, zda se jedná o kolekci uloženou v paměti, v XML souboru nebo třeba v databázi), delegáty a události (obdoba ukazatelů na funkce jazyka C/C++), lambda výrazy a mnoho dalších. Hlavní nevýhodou oproti jazyku Java je závislost na operačním systému Windows, který je od stejného výrobce.

4.1.2 Windows Forms aplikace

Techonologie pro vývoj uživatelského rozhraní pro Windows aplikace vytvořena firmou Microsoft. Aplikace je řízena na základě událostí, například po stisknutí určitého tlačítka se vykoná určitá metoda. Vývoj se provádí ve dvou prostředích. První se nazývá designér a slouží pro grafické umísťování ovládacích prvků do okna aplikace. Druhé prostředí představuje klasické prostředí pro psání zdrojového kódu. Alternativou je WPF aplikace, která je podstatně mladší, nejedná se však o nástupce, ale spíše doplnění o moderní prvky, jako jsou animace.



Obrázek 9: Třídní diagram knihovní komponenty Design Patterns

4.1.3 RTF

Rich Text Format je formát vyvořen firmou Microsoft v roce 1987 pro zobrazování a ukládání dokumentů [5]. Textová data se dají formátovat pomocí příkazů, např. tučné písmo, barva, velikost, zarovnání písma apod. Tento formát je multiplatformní, takže dokumenty uložené v tomto formátu lze přečíst i na zařízeních s jiným operačním systémem.

4.1.4 XML

XML je obecný značkovací jazyk vytvořený konsorciem W3C pro snadné uchovávání a přenos dat nebo vytvoření konkrétních značkovacích jazyků. Je nezávislý na platformě nebo typu uchovávaných dat. Výhodou je snadná čitelnost a přenositelnost mezi programy. Nevýhodou oproti binárním formátům však může být velikost a rychlost serializace a deserializace. [5]

4.2 Struktura aplikace

Celé řešení obsahuje 2 komponenty - knihovnu DesignPatterns poskytující třídy pro práci s daty a Windows Forms aplikaci představující uživatelské rozhraní.

Knihovna DesignPatterns se skládá ze tří tříd (viz obr. 9). Základní třídou je DessignPattern představující návrhový vzor. Obsahuje nezbytné proměnné a vlastnosti pro zachování struktury vzoru popsané v podkapitole 2.4. Dále obsahuje metody pro převedení celého textu návrhového vzoru na formát RTF, který se používá pro zobrazování vzorů. Druhá třída DesignPatterns obsahuje kolekci všech načtených návrhových vzorů a me-

tody pro přístup k ní. Celá tato třída je statická, takže není potřeba uchovávat si instanci. Na tuto třídu by se také dal použít návrhový vzor Singleton 3.1. Nemělo by to ale moc význam, protože třída je relativně jednoduchá a neobsahuje složité kompozice. Navíc není určená pro rozšiřování. Poslední třídou je XMLSerializator. Také se jedná o třídu statickou a obsahuje metody pro serializaci a deserializaci návrhových vzorů.

Samotná Windows Forms komponenta obsahuje třídy představující jednotlivá okna aplikace. Tyto třídy jsou rozděleny na dvě části pro oddělení deklarace a definice ovládacích prvků a nastavení aplikace automaticky vygenerované designérem a implementaci událostí psaných programátorem. Hlavní okno představuje třída MainWindow. Třída NewPatternWindow poskytuje prostředí pro přidávání nových vzorů nebo jejich editaci. Třída AdvancedSearchForm představuje okno pro rozšířené vyhledávání v dostupných návrhových vzorech. Třída SpecificFilesForm dává rozhraní pro výběr mezi vzory při importu nebo exportu vzorů. Můžeme si tak vybrat jen ty vzory, které potřebujeme. Poslední třídou je SettingsWindow, která poskytuje prvky pro nastavení aplikace.

Dále jsou používány vlastní ovládací prvky „SearchPanel“ pro poskytnutí rozhraní pro zadání vstupních dat rozšířeného vyhledávání a „TextBox2“, který poskytuje rozšířené funkce pro editaci textu, například funkce odsazování textu tabulátorem, nové řádky apod.

4.3 Implementační detaily

Na začátku kapitoly 4 jsem uvedl cíle, které tato aplikace bude splňovat. Nyní uvedu detailnější popis, jak jsem těchto cílů dosáhl.

4.3.1 Jednoduché používání

Jednoduchého používaní je dosaženo interaktivnosti prvků. Snažil jsem se používání oken pro zprávy (tzv. message boxy) omezit na minimum, což jsem nahradil dynamikou aplikace. Např. tlačítka pro potvrzení jsou použitelné pouze pokud jsou nastaveny všechny potřebné hodnoty. Aplikace také reaguje na známé klávesové zkratky, např. ctrl+O pro otevření souboru se vzory, ctrl+N pro zobrazení okna pro vytvoření nového návrhového vzoru atd.

Ke každému návrhovému vzoru se také dají psát vlastní poznámky přímo v hlavním okně aplikace. Tyto poznámky se ukládají automaticky, což šetří čas uživateli při používání této aplikace.

4.3.2 Prostředky pro vytvoření, editaci a mazání vzorů

Vytvoření, popř. editace návrhového vzoru se provádí ve zvláštním okně. Jednotlivé sekce jsou rozděleny na panely. Mazání vzorů jde přímo v hlavním okně. Aplikace se před smazáním zeptá pro potvrzení. Nutno také poznamenat, že smazaný vzor již nejde obnovit zpátky.

Pro editaci vzoru nebo vytvoření nového vzoru se používá stejná třída NewPatternWindow, která vytvoří nové okno pro tyto účely. Obsahuje dva konstruktory – bezpara-

metrický pro vytvoření okna pro nový vzor a parametrický, který přijímá jako parametr návrhový vzor, který se bude editovat.

4.3.3 Možnost ukládání a načítání vzorů ze souboru

Zajištěno pomocí serializace a deserializace do XML souborů. Vzory lze ukládat jak jednotlivě, tak i ve více množství do jediného souboru nebo do zvláštních souborů v určité složce.

Před uložením nebo načtením vzoru se zobrazí okno, ve kterém lze vybrat jednotlivé vzory. Toto okno představuje třída SpecificFilesForm, která má, stejně jako třída NewPatternWindow, dva konstruktory.

Bezparametrický konstruktor vytvoří okno pro exportování vzorů. Zobrazí seznam všech dostupných vzorů, ze kterých si uživatel může vybrat, které vzory chce uložit. Dále také vytvoří prvky, kde se dá zadat cesta pro ukládání vzorů. Uživatel může přepínat mezi možností, kdy se všechny vybrané vzory exportují do jediného XML souboru a možností, kdy se každý vzor uloží samostatně do zvláštního souboru do určité složky.

Druhý konstruktor přebírá jako parametr kolekci cest k importovaným souborům. Tyto cesty určuje uživatel např. při přetáhnutí souborů pomocí funkce „Drag and drop“ nebo zadáním cesty v dialogovém okně pro výběr souborů. Následně se pokusí návrhové vzory ze zadaných souborů přečíst a zobrazí je v seznamu, kde si uživatel může vybrat, které vzory chce naimportovat.

4.3.4 Snadné sdílení návrhových vzorů

Návrhové vzory se dají importovat a exportovat do XML souborů, které se dají přenášet na jiná zařízení. Uživatel také může vybrat pouze vzory, které chce importovat nebo exportovat. Aplikace navíc podporuje funkci *drag and drop*, pomocí které se dají návrhové vzory snadno importovat přetáhnutím souborů do aplikace.

4.3.5 Rozšířené fulltextové vyhledávání v libovolné kombinaci různých sekcí

V aplikaci se dájí vzory vyhledávat fulltextově v okně „rozšířená vyhledávání“. Vyhledávání nerozlišuje malá a velká písmena. Využívám vlastního ovládacího prvku „SearchPanel“ pro zadávání vstupních hodnot. Tyto prvky se dají libovolně přidávat a odebírat, čímž uživatel může měnit počet úrovní vyhledávání.

Celé vyhledávání je prováděno pomocí dynamických lambda výrazů. Nejprve získáme z každého ovládacího prvku typ sekce, ve které se bude vyhledávat, hledanou hodnotu a pokud je více úrovní, tak také operátor představující logickou spojku „a“ nebo „nebo“. Lambda výraz se postupně vytváří pomocí reflexe na nezbytné metody objektu string „ToLower“ a „Contains“ a také na získání vlastnosti návrhového vzoru, které určuje sekci, ve které se provádí vyhledávání. Obecný lambda výraz pro celé vyhledávání se dá napsat jako:

$$vzor \Rightarrow podvyraz_1(operator_1)podvyraz_2 \ldots (operator_n)podvyraz_n$$

Podvýrazy pak mají tvar: „vzor.Vlastnost.ToLower().Contains(hodnota.ToLower())“. Metoda ToLower zajistí, že vyhledávání nebude rozlišovat mezi malými a velkými písmeny. Metoda Contains vrátí logickou hodnotu podle toho, zda vzor obsahuje v zadané sekci zadanou hodnotu. Operátor představuje logickou spojku „a“ nebo „nebo“. Každý podvýraz představuje jednu úroveň vyhledávání (počet hledaných hodnot). Nakonec se celý lambda výraz vyhodnotí. Nejprve je však nutné získat kolekci všech vzorů a převést jí na typ IQueryable, aby se dalo s touto kolekcí dále pracovat. Na tuto kolekci se poté zavolá metoda Where, která obsahuje jako parametr sestavený lambda výraz. Tato metoda následně vrátí kolekci všech návrhových vzorů, které splňují podmínky dané lambda výrazem.

4.3.6 Podpora více jazyků

Všechna okna mají nastavenou vlastnost Localizable na hodnotu true, čímž se zapne podpora aplikace pro více kultur. Kulturou můžeme chápat jiné popisy prvků, jakými jsou nápisy nebo texty tlačítek, formátování dat a časových údajů, názvy pro dny v týdnů, měsíců apod. Každé okno má několik resource souborů, každé pro jednu kulturu. Jedná se o speciální XML soubor, který obsahuje všechny data potřebné pro změnu kultury. V aplikaci jsou nastaveny dvě kultury—Česká a anglická. Přepínáním mezi němi tak můžeme měnit jazyk aplikace.

Změna kultury ve Windows Forms aplikacích za běhu je však celkem problematická pro již vytvořená okna. Jediný způsob, jak tohoto dosáhnout je projít po jednom všechny prvky v okně a změnit tak jejich vlastnosti, které potřebujeme. K tomu je určena statická třída RuntimeLocalizer obsahující potřebné metody pro změnu kultury.

Samotné přepínání kultur se dá provádět pomocí tlačítek, zobrazené jako vlajky, v pravém horním rohu hlavního okna nebo také v okně pro nastavení aplikace.

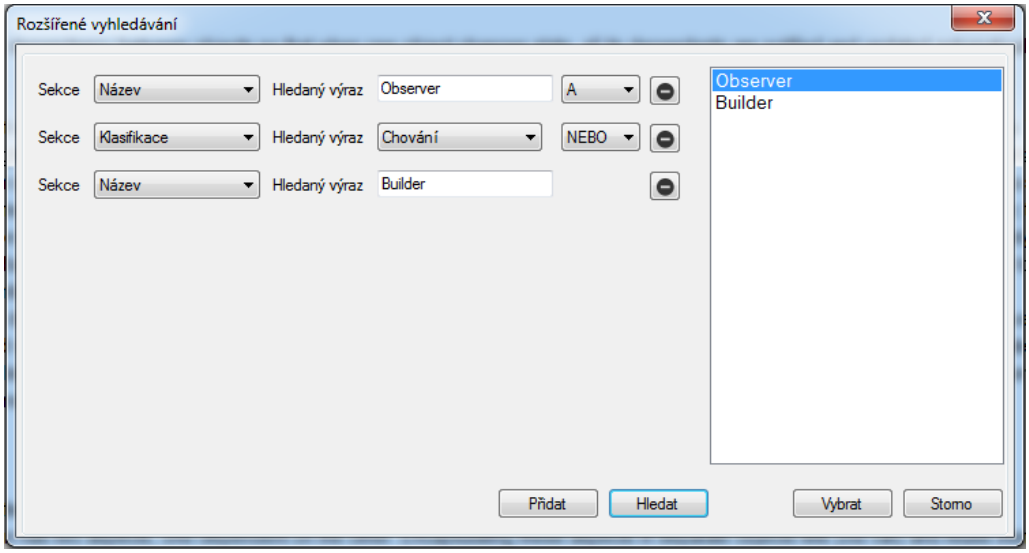
Nastavení kultura se navíc ukládá do konfiguračního souboru, takže se toto nastavení zachová i při příštím spuštění.

4.4 Testování aplikace

V této podkapitole se budu věnovat testování aplikace, především funkci rozšířeného vyhledávání vzorů. Jako testovací data jsou použité vzory z knihy Design Patterns: Elements of Reusable Object-Oriented Software[2].

4.4.1 Test č. 1

První test je vytvořen pro účely otestování funkce logických spojek.

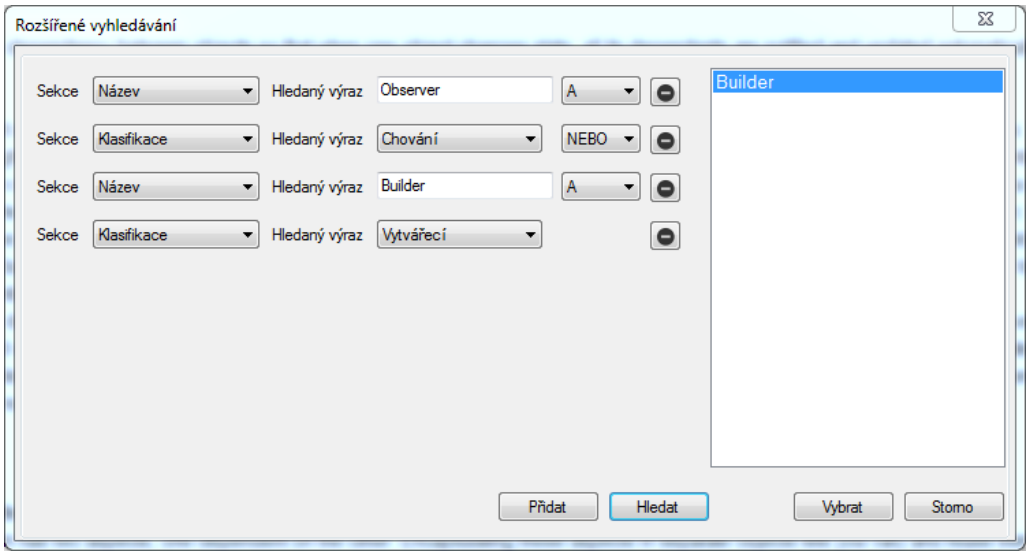


Obrázek 10: Výsledek testu č. 1

Program v tomto případě vyhledal správně výsledky podle pravidel booleovy algebry.

4.4.2 Test č. 2

V tomto testu navazuji na test č. 1. Zadáním bude složitější funkce pro otestování priority operátoru „a“ před operátorem „nebo“.

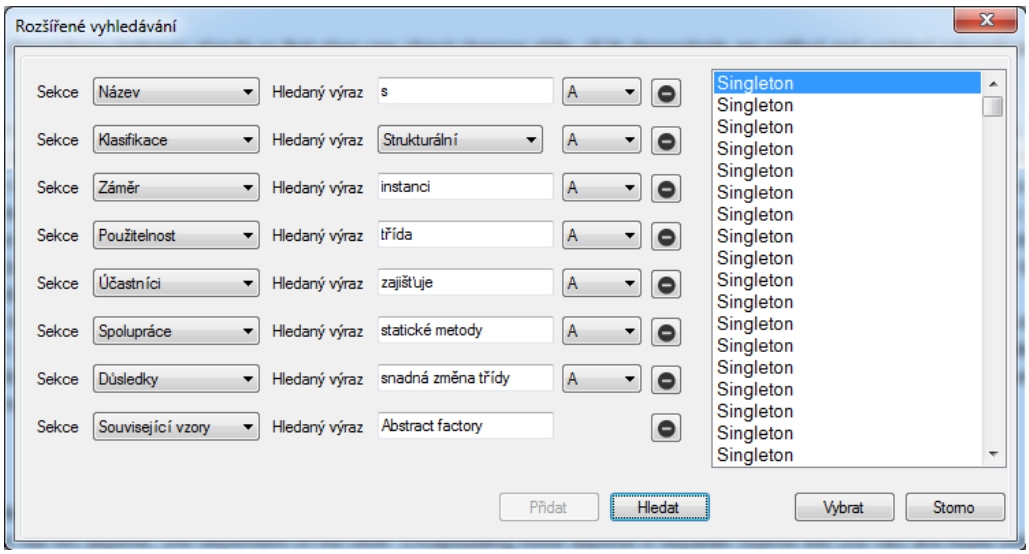


Obrázek 11: Výsledek testu č. 2

Program v tomto případě vrátil pouze jeden vzor, i když podle funkce $y = a * a + b * b$ by měl vrátit dva návrhové vzory. Z pozorování jsem došel k závěru, že priorita operátoru „a“ zde neplatí. Funkce se tak dá znázornit jako $y = ((a * a) + b) * b$.

4.4.3 Test č. 3

Třetí test je prováděn pro otestování rychlosti vyhledávání při použití hledání s maximálním počtem vyhledávacích úrovní s vysokým počtem návrhových vzorů. Rozmnožil jsem mnohonásobně návrhové vzory, takže výsledný počet vzorů pro testovací účely je 5400, což je zcela dostačující, protože reálný odhad maximálního počtu vzorů může být pouze v řádech stovek. XML soubor s těmito vzory má velikost 47MB, což je také vyhovující.

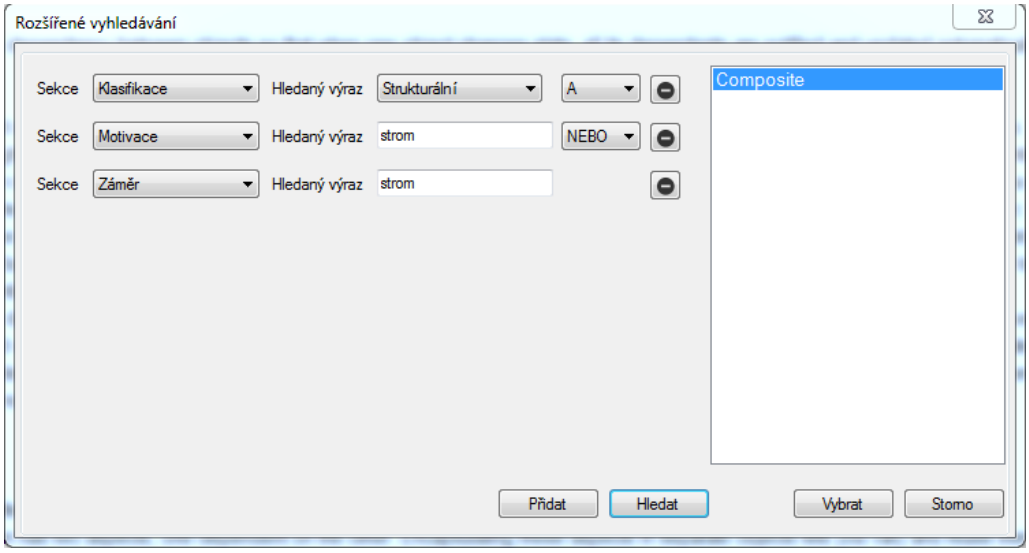


Obrázek 12: Výsledek testu č. 3

Serializace a deserializace probíhá bez problémů a na rychlosti se to neprojevuje. Zadávané hodnoty jsou takové, aby výsledek vrátil pouze jeden návrhový vzor. Při vyhledávání nebylo vysoké množství dat poznat. Z hlediska rychlosti zpracování dat je aplikace optimální.

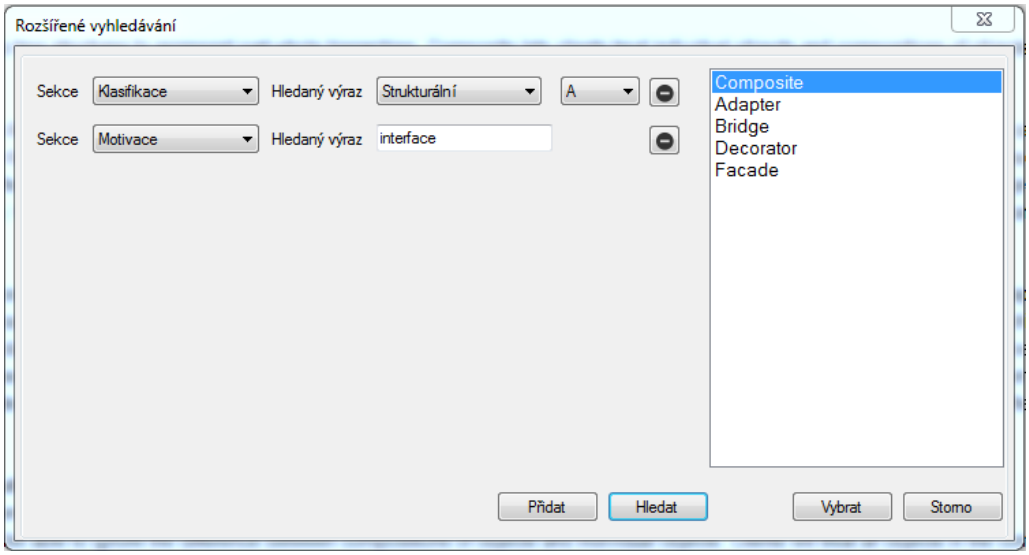
4.4.4 Test č. 4

Testem č. 4 otestuji vyhledávání používáním běžným uživatelem a zároveň demonstruji přínos rozšířeného vyhledávání. Zadáním je situace, kdy uživatel pouze ví, pro jakou situaci vzor potřebuje. Při návrhu vlastního softwarového systému se naskytl problém, kdy je potřeba strukturovat množství podobných objektů do stromové struktury. Aplikace tedy bude hledat všechny strukturální návrhové vzory s klíčovým slovem „strom“ v sekci motivace nebo záměr.



Obrázek 13: Výsledek testu č. 4a

Aplikace v tomto případě vyhledala návrhový vzor Composite, který odpovídá zadání. Ne vždy však aplikace vrátí pouze jeden výsledek. Příkladem může být situace, kdy uživatel potřebuje strukturovat mnnožství podobných objektů a poskytnout jim společné rozhraní. Aplikace tak bude hledat všechny strukturální návrhové vzory, které obsahují v sekci motivace slovo „interface (rozhranní)“.



Obrázek 14: Výsledek testu č. 4b

V tomto případě aplikace našla 5 návrhových vzorů. Uživatel pak může procházet jednotlivé vzory a vybrat ten, který nejvíce odpovídá zadání.

4.4.5 Vyhodnocení testů

Cílem testů bylo ověření funkce rozšířeného vyhledávání návrhových vzorů. Prvními dvěma testy jsem otestoval prioritu logický spojek při skládání dotazů. Ukázalo se, že priorita spojky „a“ oproti spojce „nebo“ při vyhledávání neplatí. Třetí test byl zátěžový. Program pracoval s 5400 návrhovými vzory, což převyšuje reálný odhad počtu vzorů při běžném provozu více než 50 krát. Načítání a ukládání těchto vzorů bylo velice rychlé navzdory velikosti dat v XML souboru stejně jako následný test vyhledávání s maximálním počtem hledaných úrovní. Posledním testem bylo vyzkoušení vyhledávání vzorů na základě okolností, které se mohou naskytnout při vývoji softwarového systému. Ukázal jsem příklady dvou situací, kdy v první situaci aplikace vyhledalo jeden konkrétní vzor. Ve druhé situaci však bylo návrhových vzorů vyhledáno více. V tomto případě rozšířené vyhledávání pomáhá k filtraci vzorů a tedy k následnému výběru.

5 Závěr

Cílem této práce bylo nastínit problematiku dokumentace návrhových vzorů v softwarovém inženýrství a vyhledávání v katalogu mezi němi. Práce je rozdělena na dvě části. První část je teoretická, ve které byl popsán vznik a přínos návrhových vzorů, struktura a způsob dokumentace. Dále byly ukázány konkrétní příklady návrhových vzorů. Pro účely dokumentace a navigace v těchto vzorech byla vytvořena aplikace. Popisem a testováním této aplikace se zabývala druhá část práce.

Při psaní této bakalářské práce jsem lépe pochopil strukturování vzorů a jejich celkový přínos do vývoje softwarových systémů. Také se dokážu mezi těmito vzory lépe orientovat. Během implementace jsem objevil spoustu nových pokročilejších nástrojů a technik pro vývoj Windows Forms aplikací a práci s daty obecně.

Aplikace by se dala do budoucna rozšířit o nové funkce a nástroje, například sdílení návrhových vzorů na webových stránkách, detekce návrhových vzorů ve zdrojových kódech nebo generování základních prvků návrhových vzorů ve formě zdrojových kódů přímo v této aplikaci. Aplikace by tak mohla poskytovat profesionální nástroje pro práci s návrhovými vzory, které jsou v dnešní době velice potřebné. Návrh a implementace takových nástrojů by však bylo v rozsahu několika dalších prací.

6 Reference

[1] ALEXANDER, Christopher, Sara ISHIKAWA a Murray SILVERSTEIN *A Pattern Language: Towns, Buildings. Construction*, New York: Oxford University Press, 1977, 1171 s. ISBN 01-950-1919-9.

[2] GAMMA, Erich, Richard HELM, Ralph JOHNSON a John VLISSIDES *Design Patterns: Elements of Reusable Object-Oriented Software*, Massachusetts: Addison-Wesley, c1995, 395 s. ISBN 02-016-3361-2.

[3] FOWLER, Martin *Patterns of Enterprise Application Architecture*, Boston: Addison-Wesley, c2003, 533 s. The Addison-Wesley Signature Series. ISBN 978-0-321-12742-6

[4] BROWN, William J. *AntiPatterns: refactoring software, architectures, and projects in crisis*, New York: Wiley, 1998, 309 s. ISBN 04-711-9713-0.

[5] Wikipedia EN zdroj: http://en.wikipedia.org

A Výpisy programů

```
public class Singleton
{
    protected Singleton instance = null;

    private Singleton()  {}

    public static Singleton GetInstance()
    {
        if (instance != null)
            instance = new Singleton();

        return instance;
    }
}
```

Výpis 1: Příklad implementace vzoru Singleton v jazyce C#

```
public class Component
{
    public Component() {}

    public virtual void Method() {}
}

public class Composite : Component
{
    private List<Component> childrens = new List<Component>();

    public Composite() {}

    public void AddChild(Component child)
    {
        childrens.Add(child);
    }

    public void RemoveChild(Component child)
    {
        childrens.Remove(child);
    }

    public virtual void Method()
    {
        foreach (var child in childrens)
        {
            child.Method();
        }
    }
}
public class Leaf : Component
{
    public Leaf() {}
```

```
    public override void Method()
    {
        // Vlastni  implementace
    }
}
```

Výpis 2: Příklad implementace vzoru Composite v jazyce C#

```
public class Subject
{
        public virtual  void Notify( List<Observer> listeners)
        {
                foreach (var  listener  in  listeners )
                {
                        listener .Update();
                }
        }
}

public class ConcreteSubject : Subject
{
        private  List<Observer> listeners = new List<Observer>();

        public void AddListener(Observer listener)
        {
                listeners .Add(listener) ;
        }

        public void RemoveLister(Observer listener)
        {
                listeners .Remove(listener);
        }
}

public class Observer
{
        public virtual  void Update() {}
}

public class ConcreteObserver : Observer
{
        public override void Update() {}
}
```

Výpis 3: Příklad implementace vzoru Observer v jazyce C#